Technical Disclosure Commons

Defensive Publications Series

May 01, 2017

Independent Partially Upgradable Mandatory Access Control System

Daniel Cashman

Independent partially upgradable mandatory access control system

ABSTRACT

Mandatory access control (MAC) is a mechanism by which an operating system provides a secure computing environment by restricting the ability of processes or threads to perform operations on objects. Objects are provided with type labels and a policy that governs how interactions between labeled objects may occur. MAC systems enforce a centralized policy that governs access to the entire system and is incompatible with operating systems where components are separately owned and upgraded.

This disclosure provides techniques to write security policies for objects that are accessed by both platform and non-platform components. Per techniques disclosed, secure access to objects within separately owned components is provided by converting labels assigned to OS objects to versioned attributes that are inherited by future labels of those objects. A mapping is created and maintained between different versions of a label. The mapping is used to ensure that security policy works across versions of objects. The techniques enable new policies to be implemented, or old policies to be changed, without affecting non-platform components, and permit non-platform components to upgrade and eventually lose deprecated rules.

KEYWORDS

- modular operating system
- mandatory access control
- object label versioning
- component upgradability

BACKGROUND

Mandatory access control (MAC) is a technique to create a secure computing environment by restricting the ability of processes or threads to perform operations on objects. MAC functionality is provided by operating systems, e.g., SELinux is a popular Linux kernel-based module that provides MAC. SELinux builds on the linux security modules (LSM) of the linux kernel to enable the definition of policy indicating how objects as defined within LSM and user-space components interact.

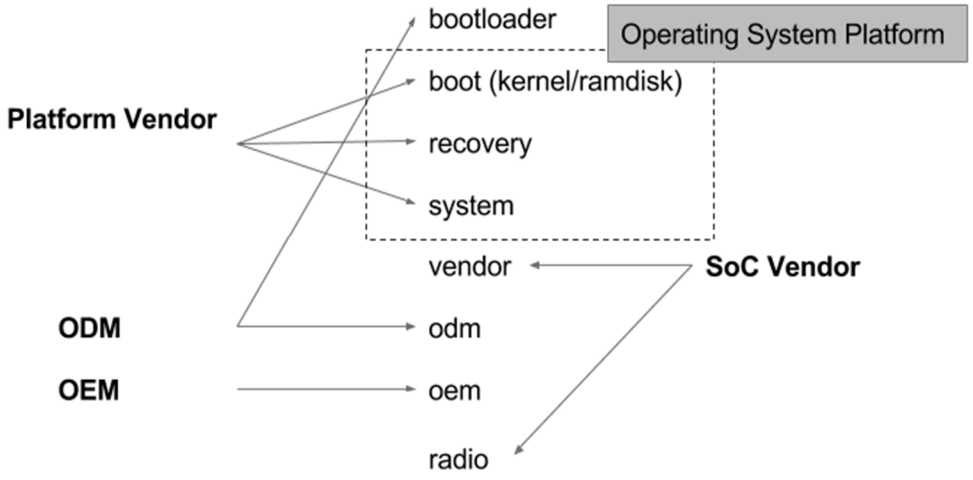

Fig. 1: Partitions of a mobile device

Fig. 1 shows example partitions of a mobile computing device, e.g., a smartphone. Certain images within the partition, e.g., “boot,” “recovery,” “system,” etc. include operating system kernel/ framework code and configurations, and are accessible by the platform vendor, e.g., a manufacturer of mobile device. Other images, e.g., “vendor,”, “radio,” etc. include code specific or proprietary to a System-on-a-Chip (SoC) vendor. Other images, e.g., “odm,” “oem,” etc., include code or configurations specific to an original device manufacturer (ODM) and original equipment manufacturer (or carrier).

Recent trends in operating system design call for a split of the OS into different components, e.g., to enable separate ownership and upgradability of individual components. As illustrated above, one component may be owned by the OS platform, while another by an ODM, yet another by the SoC vendor, etc. One goal of splitting the OS into independently upgradable components is to allow SoC vendors, ODMs, OEMs etc. to customize their components, e.g., SELinux settings, in an isolated manner without cross-partition modifications. However, the MAC model for secure computing, which requires centralized policy that governs access for the entire system, is incompatible with the splitting of the OS into independently upgradable components. At present, if the owner of one part of the system does not know the composition of another part, it is not possible to write policy that is centrally unified and applicable to all objects on a system.

SELinux attempts to address the problem of vendor-specific customization by allowing for policy modules on top of a base module. In this technique, a base reference policy provides policy common to a particular OS distribution. The base policy can be added to by adding new policy in the form of modules. However, this technique relies on the module policy writers knowing the base policy against which the modules are being written. This is not necessarily true in the case of operating systems with independently upgradable components, since the platform component can itself be upgraded independent of the non-platform components. Thus, the platform component (which is analogous to the base reference policy of SELinux) can effectively change from underneath the module writer. This can render invalid an access policy rule or label type that is relied upon by a policy module. The approach of SELinux to vendor-specific customization allows for the platform or base-policy writer to not know about device-specific policy. However, this approach doesn't allow for independent platform upgrades.

The technique of using versions for labels has been used in the past. However, versions have generally been used to target symbols in shared libraries, archives, or object files. Such version labeling applies, e.g., to code compilation and linking, rather than OS objects that each have only one non-duplicable label. Current version labeling is not compatible with the one central policy of the MAC model that requires updated platform and non-platform policies to combine to generate one functional unit, rather than simply targeting one or the other.

DESCRIPTION

In an operating system with independently upgradable components the platform has limited or no knowledge of the additional components that will be provided by individual vendors. Correspondingly, vendors do not know the future composition of an upgraded platform. Recognizing that there exist objects that are accessed by both platform and non-platform (e.g., device- or vendor-specific) components, object-access policy is written for both types of components. Such object-access policy depends on version of the labels of objects.

Techniques of this disclosure enforce, through policy and testing, the following conditions:

- The labels of an object are owned by only one of the platform policies and non-platform policies.
- A dependency on the platform by a non-platform component such that the platform can upgrade without an upgrade to the non-platform component is permitted. However, the reverse is disallowed such that non-platform policy always targets a particular platform.

Existing MAC infrastructure is modified such that the ownership is respected in accordance with the above conditions. There rules are maintained while still allowing the different policies to be combined.

Under the above conditions, techniques of this disclosure provide secure access to objects within separately owned components as follows:

- The platform MAC policy is divided into public and private components. The public component consists of the type labels and attributes of the portions of the system which the non-platform components interact with and write policy for. This is equivalent to an API definition between the platform and non-platform components.
- The public MAC policy is exported to enable its inclusion as part of the non-platform policy. At the interface between platform and non-platform components, all concrete types (labels) assigned to OS objects are converted into versioned attributes that can be inherited by future labels of those objects, whether such objects are the same or changed. In an implementation, each label has a version-attribute chain that extends back to all the versions that exist for the object it represents. This ensures that the non-platform policy can interoperate with those objects as before.
- A mapping between the exported types of one platform version and future versions is created and maintained. This enables any changes to the labels of the objects at platform-device interface to be mapped back to old, versioned labels. In an implementation, multiple different mappings that each correspond to an old version are tracked and the appropriate version is chosen during operation. In this manner, objects that are labeled with a type do not break behavior guaranteed by the public component of the policy in a previous version. Old policy continues to work upon objects even when a platform is upgraded.

Techniques of this disclosure represent objects of an OS as an API and enable versioning of object types. Compatibility is maintained across the platform-device interface via a mapping, e.g., a mapping file that tracks future development. These techniques enable new policies to be implemented or old policies to be changed with no effect on non-platform components, e.g., without breaking assumptions made by non-platform components. Implementing these techniques allows non-platform components to upgrade and eventually lose deprecated rules.

Per the techniques disclosed herein, a strict interface between platform and non-platform components is enforced by MAC while maintaining the traditional guarantees provided by the MAC, e.g., a centralized policy, even in the absence of knowledge of the entire system.

Strict rules are placed to define the platform-device interface. Implementations of these techniques may require sacrificing some policy flexibility. Further, some testing or development burden is added in order to ensure compliance with the rules and avoid future incompatibility issues. However, these burdens may be small in comparison with benefits of the techniques in providing secure computing within the context of independently upgradable OS components. Alternative solutions to the proposed techniques have their own drawbacks. For example, committing to a strict API from a policy development perspective and preventing changes to labels of interface-level objects eliminates the useful facility of MAC customization or policy tightening for the purpose of hardened security. As a further example, changing the LSM MAC infrastructure to enable objects to have two simultaneous labels may be a rather invasive change and can make the security policy less clear and less secure.

Techniques of this disclosure apply to any MAC system, including that provided by the framework of SELinux. If implemented under SELinux, versioning is done by using SELinux

attributes, specifically turning types into versioned attributes, such that new or changed types can automatically act as though these were referred to before.

CONCLUSION

Mandatory access control (MAC) within operating systems is a mechanism to provide a secure computing environment. MAC works by labeling objects and enforcing policy that governs interactions between labeled objects. The mechanism of MAC requires enforcement of a centralized policy applicable to the entire system. Centralized access policy, however, is incompatible with modular operating systems that allow independent upgrades to components. This disclosure addresses the problem of secure access to objects accessed by independently upgradable OS components. Per technique disclosed herein, a versioning system is implemented for such objects. In effect, objects of an OS are represented as an API that supports versioning of object types. To retain interoperability between components as they upgrade independently, a mapping of labels is created and maintained. This mapping is used to ensure that policy works across versions of objects.